C# 프로그래밍

- 4 주차 (2강)

SEJONG UNIVERSITY

객체지향 프로그래밍

- 은닉성 (캡슐화)
- 상속성
- 다형성

SEJONG UNIVERSITY

정보 은닉 (Information Hiding)

- 외부에서 클래스의 멤버 변수에 직접 접근 불가
  - ✓ 특별한 이유를 제외하고는 필드를 public 으로 선언하지 않음
  - ✓ 접근이 필요할 때는 접근자(getter)/설정자(setter) 메소드 이용해 외부 접근을 관리

접근자와 설정자 정의

```
class Circle
{
    double pi = 3.14;

    public double GetPi()
    { // 접근자
        return pi;
    }

    public void SetPi(double value)
    { // 설정자
        pi = value;
    }
}
```

설정자 통해 외부 입력 관리

```
class Circle
{
    public void SetPi(double value)
    { // 설정자
        if (value <= 3 || value >= 3.15)
        {
            Console.WriteLine("문제 발생");
        }
        pi = value;
    }
    // ...
}
```

```
static void Main(string[] args)
{
    Circle o = new Circle();
    o.SetPi(3.14159);
    o.SetPi(3.5); // 출력: 문제 발생
}
```

프로퍼티 (Property)

- C# 에서는 접근자/설정자를 쉽게 정의하고 사용하도록 프로퍼티 문법 제공
  - ✓ 설정자 set 의 암묵적 매개변수로 "value" 예약어 사용

프로퍼티 사용 형식

```
class 클래스이름
{
    데이터형식 필드이름;
    접근제한자 데이터형식 프로퍼티이름
    {
        get
        {
            return 필드이름;
        }

        set
        {
            필드이름 = value;
        }
    }
}
```

프로퍼티 작성 예제

```
class Circle
{
    double pi = 3.14;
    public double Pi
    {
        get { return pi; }
        set { pi = value; }
    }
}
```

```
    double pi = 3.14;
    public void set_Pi(double value)
    {
        this.pi = value;
    }
    public double get_Pi()
    {
        return this.pi;
    }
```

프로퍼티 쓰기/읽기

```
static void Main(string[] args)
{
    Circle o = new Circle();
    o.Pi = 3.14159; // set
    double piValue = o.Pi; // get
}
```

프로퍼티 코드는 컴파일러에 의해 빌드 시점에서 변환

프로퍼티 예제 코드

```
class BirthdayInfo
{
    private string name;
    private DateTime birthday;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public DateTime Birthday
    {
        get { return birthday; }
        set { birthday = value; }
    }

    public int Age  // 읽기 전용 프로퍼티
    {
        get { return new DateTime(DateTime.Now.Subtract(birthday).Ticks).Year; }
    }
}
```

```
static void Main(string[] args)
{
    BirthdayInfo birth = new BirthdayInfo();
    birth.Name = "홍길동";
    birth.Birthday = new DateTime(1991, 6, 28);
    Console.WriteLine($"Name : {birth.Name}");
    Console.WriteLine($"Birth : {birth.Birthday.ToShortDateString()}");
    Console.WriteLine($"Age : {birth.Age}");
}
```

출력 결과:

```
Name : 홍길동
Birth : 1991-06-28
Age : 30
```

자동구현 프로퍼티

- C# 3.0부터 단순히 필드를 읽고 쓰기만 할 때 자동구현 프로퍼티 사용 가능
  - ✓ C# 7.0 부터는 자동구현 프로퍼티 선언과 동시에 초기화 수행 가능

일반 프로퍼티 사용

```
public class NameCard
{
    private string name;
    private string phoneNumber;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string PhoneNumber
    {
        get { return phoneNumber; }
        set { phoneNumber = value; }
    }
}
```

자동구현 프로퍼티와 초기화

```
public class NameCard
{
    public string Name { get; set; } = "Unknown";
    public string PhoneNumber { get; set; } = "000-0000";
}
```

자동 구현 프로퍼티 예제 코드

```
class BirthdayInfo
{
    public string Name { get; set; } = "Unknown";
    public DateTime Birthday { get; set; } = new DateTime(1, 1, 1);
    public int Age  // 읽기 전용 프로퍼티
    {
        get { return new DateTime(DateTime.Now.Subtract(Birthday).Ticks).Year; }
    }
}
```

```
static void Main(string[] args)
{
    BirthdayInfo birth = new BirthdayInfo();
    Console.WriteLine($"Name : {birth.Name}");
    Console.WriteLine($"Birth : {birth.Birthday.ToShortDateString()}");
    Console.WriteLine($"Age : {birth.Age}");

    birth.Name = "홍길동";
    birth.Birthday = new DateTime(1991, 6, 28);

    Console.WriteLine($"Name : {birth.Name}");
    Console.WriteLine($"Birth : {birth.Birthday.ToShortDateString()}");
    Console.WriteLine($"Age : {birth.Age}");
}
```

출력 결과:

```
Name : Unknown
Birth : 0001-01-01
Age : 2021
Name : 홍길동
Birth : 1991-06-28
Age : 30
```

프로퍼티와 생성자

- 객체를 생성할 때 프로퍼티를 이용해 각 필드를 초기화
  - ✓ <프로퍼티 = 값> 목록에 객체의 모든 프로퍼티가 올 필요는 없음
  - ✓ 초기화하고 싶은 프로퍼티만 넣어서 초기화

프로퍼티를 이용한 초기화 형식

```
클래스이름 인스턴스 = new 클래스이름()
{
    프로퍼티1 = 값,
    프로퍼티2 = 값, // 콤마(,)로 구분
    프로퍼트3 = 값
};
```

객체 생성할 때 프로퍼티를 이용한 초기화

```
BirthdayInfo birth = new BirthdayInfo()
{
    Name = "홍길동",
    Birthday = new DateTime(1991, 6, 28)
};
```

프로퍼티와 생성자 예제 코드

```
class BirthdayInfo
{
    public string Name { get; set; } = "Unknown";
    public DateTime Birthday { get; set; } = new DateTime(1, 1, 1);
    public int Age  // 읽기 전용 프로퍼티
    {
        get { return new DateTime(DateTime.Now.Subtract(Birthday).Ticks).Year; }
    }
}
```

```
static void Main(string[] args)
{
    BirthdayInfo birth = new BirthdayInfo()
    {
        Name = "홍길동",
        Birthday = new DateTime(1991, 6, 28)
    };

    Console.WriteLine($"Name : {birth.Name}");
    Console.WriteLine($"Birth : {birth.Birthday.ToShortDateString()}");
    Console.WriteLine($"Age : {birth.Age}");
}
```

출력 결과:

```
Name : 홍길동
Birth : 1991-06-28
Age : 30
```

초기화 전용 자동 구현 프로퍼티

- 프로퍼티를 객체를 생성할 때 초기화 후, 중간에 변경 못 하도록 설정
  - ✓ 자동 구현 프로퍼티에서 set 키워드 대신에 init 키워드 사용

생성자로 초기화 후 get 키워드만 사용

```
class Transaction
{
    public Transaction(string _from, string _to, int _amount)
    {
        from = _from; to = _to; amount = _amount;
    }

    string from;
    string to;
    int amount;

    public string From { get { return from; } }
    public string To { get { return to; } }
    public int Amount { get { return amount; } }
}
```

초기화 전용 자동구현 프로퍼티

```
class Transaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

초기화 전용 자동 구현 프로퍼티 예제 코드

```
class Transaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

출력 결과:

```
Alice       -> Bob         : 100
Bob         -> Charlie     : 50
Charlie     -> Alice       : 50
```

```
static void Main(string[] args)
{
    Transaction tr1 = new Transaction { From = "Alice", To = "Bob", Amount = 100 };
    Transaction tr2 = new Transaction { From = "Bob", To = "Charlie", Amount = 50 };
    Transaction tr3 = new Transaction { From = "Charlie", To = "Alice", Amount = 50 };

    // tr1.From = "Charlie";    값 할당 시 컴파일 에러 발생

    Console.WriteLine($"{tr1.From, -10} -> {tr1.To,-10} : {tr1.Amount,-10}");
    Console.WriteLine($"{tr2.From,-10} -> {tr2.To,-10} : {tr2.Amount,-10}");
    Console.WriteLine($"{tr3.From,-10} -> {tr3.To,-10} : {tr3.Amount,-10}");
}
```

레코드 형식의 불변 객체

- 클래스는 참조 형식이기 때문에 객체 사이의 필드 복사, 비교, 출력 등에 있어 추가적 구현 필요
- record 키워드를 사용하는 레코드 형식 사용
  - ✓ 값을 담는 용도의 클래스의 역할
  - ✓ 컴파일 시 복사, 비교, 출력 등의 메서드 자동추가
  - ✓ 따라서, record 형식은 class + "기본 생성 코드"
- 레코드 형식의 불변 객체

```
record RTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

객체 생성 및 필드 초기화

```
RTransaction tr1 = new RTransaction { From = "Alice", To = "Bob", Amount = 100 };
RTransaction tr2 = new RTransaction { From = "Bob", To = "Charlie", Amount = 50 };
```

레코드 형식의 불변 객체 예제 코드

```
record RTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

출력 결과:

```
Alice       -> Bob         : 100
Bob         -> Charlie     : 50
```

```
static void Main(string[] args)
{
    RTransaction tr1 = new RTransaction { From = "Alice", To = "Bob", Amount = 100 };
    RTransaction tr2 = new RTransaction { From = "Bob", To = "Charlie", Amount = 50 };

    Console.WriteLine($"{tr1.From,-10} -> {tr1.To,-10} : {tr1.Amount,-10}");
    Console.WriteLine($"{tr2.From,-10} -> {tr2.To,-10} : {tr2.Amount,-10}");
}
```

with 를 이용한 레코드 복사

- with 키워드를 사용해 두 record 객체 사이의 깊은 복사를 수행
  - ✓ 깊은 복사와 동시에 일부 필드 값 변경 가능

```
record RTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

출력 결과:

```
Alice       -> Bob         : 100
Alice       -> Charlie     : 100
Charlie     -> Bob         : 50
```

```
static void Main(string[] args)
{
    RTransaction tr1 = new RTransaction { From = "Alice", To = "Bob", Amount = 100 };
    RTransaction tr2 = tr1 with { To = "Charlie" };     // with 사용을 통한 깊은 복사
    RTransaction tr3 = tr1 with { From = "Charlie", Amount = 50 };  // 일부 필드 값 수정

    Console.WriteLine($"{tr1.From,-10} -> {tr1.To,-10} : {tr1.Amount,-10}");
    Console.WriteLine($"{tr2.From,-10} -> {tr2.To,-10} : {tr2.Amount,-10}");
    Console.WriteLine($"{tr3.From,-10} -> {tr3.To,-10} : {tr3.Amount,-10}");
}
```

레코드 객체 비교하기

- 클래스에서는 자신과 다른 객체를 비교하기 위해 Equals() 메소드 사용
  - ✓ 필드들을 일일이 비교하기 위해 Equals() 메소드 재정의 (override) 필요
  - ✓ 재정의 없이 Equals() 메소드 사용 시 객체의 참조 주소 값만 비교
- 레코드 객체는 컴파일러가 Equals() 메소드를 자동으로 구현

```
class CTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }

    public override bool Equals(object obj)
    {
        CTransaction target = (CTransaction)obj;
        if (this.From == target.From && this.To == target.To && this.Amount == target.Amount)
            return true;
        else
            return false;
    }
}
```

```
record RTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

레코드 객체 비교 예제 코드 (1)

```
class CTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }

    public override bool Equals(object obj)
    {
        CTransaction target = (CTransaction)obj;
        if (this.From == target.From && this.To == target.To && this.Amount == target.Amount)
            return true;
        else
            return false;
    }
}
```

```
record RTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

```
static void Main(string[] args)
{
    CTransaction trA = new CTransaction { From = "Alice", To = "Bob", Amount = 100 };
    CTransaction trB = new CTransaction { From = "Alice", To = "Bob", Amount = 100 };
    Console.WriteLine(trA.Equals(trB));

    RTransaction tr1 = new RTransaction { From = "Alice", To = "Bob", Amount = 100 };
    RTransaction tr2 = new RTransaction { From = "Alice", To = "Bob", Amount = 100 };
    Console.WriteLine(tr1.Equals(tr2));
}
```

출력 결과:

```
True
True
```

레코드 객체 비교 예제 코드 (2)

```
class CTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

```
record RTransaction
{
    public string From { get; init; }
    public string To { get; init; }
    public int Amount { get; init; }
}
```

출력 결과:

```
False
True
```

```
static void Main(string[] args)
{
    CTransaction trA = new CTransaction { From = "Alice", To = "Bob", Amount = 100 };
    CTransaction trB = new CTransaction { From = "Alice", To = "Bob", Amount = 100 };
    Console.WriteLine(trA.Equals(trB));

    RTransaction tr1 = new RTransaction { From = "Alice", To = "Bob", Amount = 100 };
    RTransaction tr2 = new RTransaction { From = "Alice", To = "Bob", Amount = 100 };
    Console.WriteLine(tr1.Equals(tr2));
}
```

무명 형식

- 형식의 선언과 동시에 객체를 할당
  - ✓ 객체를 만들고 다시는 그 형식을 사용하지 않을 때 사용
  - ✓ 무명 형식의 프로퍼티에 할당된 값은 변경 불가 (읽기만 가능)

일반적인 데이터 형식은 이름이 있음

```
int a;
double b;
string c;
MyClass d = new MyClass();
```

무명 형식의 사용 예
- 괄호 { 와 } 사이에 임의의 프로퍼티 이름을 적고 값을 할당

```
var a = new { Name = "홍길동", Age = 123 };
```

무명 형식의 객체는 여느 객체처럼 프로퍼티에 접근 가능

```
Console.WriteLine($"Name: {a.Name}, Age: {a.Age}");
```

무명형식 예제 코드

```
static void Main(string[] args)
{
    var a = new { Name = "홍길동", Age = 123 };
    Console.WriteLine($"Name: {a.Name}, Age: {a.Age}");

    var b = new { Subject = "수학", Scores = new int[] { 90, 80, 70, 60 } };
    Console.Write($"Subject: {b.Subject}, Scores: ");

    foreach (var score in b.Scores)
        Console.Write($"{score} ");
    Console.WriteLine();
}
```

출력 결과:

```
Name: 홍길동, Age: 123
Subject: 수학, Scores: 90 80 70 60
```

Thank you!